windows assembler lesson

a key assembler lesson

republished 2010, here small but: full download with all samples, etc, complete

some wild basics

why another assembler publication

Back in the old days, it was a86, a text editor and a lot of hard hacking, and you got it made, eventually: you could understand and code yourself in assembler.

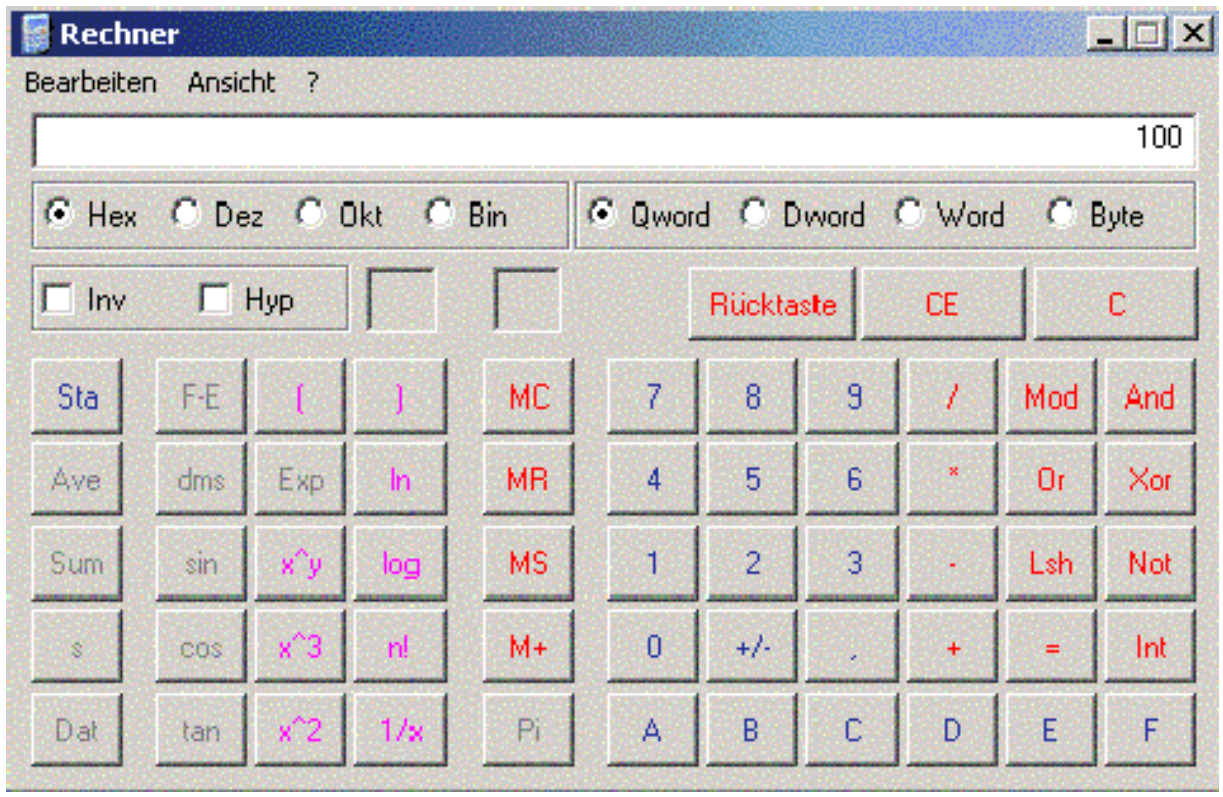
Now we do have a lot more publications, and it is nearly impossible, to find yourself a way to understand what is going on beneath the surface.

Especially when you meet an unexpected error in your well known debugger of whatever IDE you are programming, there is no way out if you want to know what is going on: you have to learn program assembler, rather learn windows assembler: masm32, flatassembler, gobug, a386 or whatever the assembly language dialect, essentially you have to learn windows assembler. I was very familiar with assembler in DOS, but it was close to impossible, to find a good lead into it for win32, except, of course, studying examples of simple programs.

The following is what I did wish I had found, when I decided to study assembler for windows.

no need to install dos to learn this

In DOS, a .com program was as straight forward matter: the first 0x100 byte were reserved for the file header, and the command at 0x100 was the first one executed. (Note, 0x100 is the same as 256, easiest way to verify this: open your windows calculator, change the view to scientific, click hex, type 100, and then dec)



pdfcrowd

As we start counting with 0, we have 256 positions, and 0x100 is position 257 - same as with numbering in a string. (e.g. in c char string[256], the highest string position is 255, like string[255] = 'x'; but string[256] = 'y'; would not be possible)

So if you would create any file, in DOS, even rundos00.txt with notepad, and rename it to rundos00.com and then make sure, position 257, when run in DOS is understood by DOS as a proper assembler command, you would have a simple, but actually running assembler program, which we will do here later. (Not anything big, but running, nevertheless).

And yes, you can program assembler using a text editor only. A most simple assembler command is 'NOP', no operation, which has the length of one byte, 0x90. To get this into a text file, you just have to hold the alt key down and type with the numeric key pad 0144, and you have 0x90 in the text file. Why exactly 0144 - to verify with the calculator. And why 'NOP' is 0x90, perhaps best to compare in one of the many documentation from masm32.But this is not our running program jet.

There are two things to consider: where exactly is position 257, and second, the so called program header, especially as seen from DOS viewpoint.

A note aside, the samples here will be reproducible from your windows xp or 2k, but most of the mechanical details are easier demonstrated from the DOS console, and the cmd prompt and the utilities offered from my site, is all we need. If you did never open a cmp prompt, please look this up in your windows help. A very handy way is available with 'More Properties' utility, which, if you set it, makes is possible to open a command prompt for any directory with right mouse click, even if it is not programmed for 2k and xp. You can also use MCL, the keyboard applet from mlin.net.

program header for a dos .com

The program header of a .com file is created by DOS the moment a .com program is started. How can we know this? Let us make a text file, rundos00.txt, type perhaps about 20 'P' into it, save the file and rename it rundos00.com. If we would have a full DOS 3.3 installation and copy it there, we could type now, at the DOS prompt, (but within the right directory) debug rundos00.com, and then (for unassemble) u0100, we would see something like this:

```
0100 50                PUSH    AX
0101 50                PUSH    AX
0102 50                PUSH    AX
0103 50                PUSH    AX
0104 50                PUSH    AX
0105 50                PUSH    AX
0106 50                PUSH    AX
0107 50                PUSH    AX
```

There is a debugger, which was quite famous back then, which still works from xp cmd prompt: Advanced File Debugger, (Sorry if I might break any copyright when offering AFD here for download, please let me know, and I will comply.) (It is kind of a nasty bastard, if you know only your mouse, but the following few hints should make AFD usable. Type quit at AFD's CMD prompt to close it. Type 'l' and a space and your path including full file name to load, e.g. nop.com - if you did rename your textfile with 0144. w and enter to save a file, such loaded and possibly changed. F4 without anything and PageDown will scroll the help, and 'a' with 0100 and enter will make your NOP editable, Ctrl+Enter gets you out of the assemble mode.)

But you do not need to do any of this to understand this lecture

Back to rundos00.com, we would see push ax as many times as we did type the 'P'. How come?

pdfcrowd

DOS understood the rundos00.com as a proper file to execute, did add the header to our rundos00.com, which is exactly 0x0100 bytes long, and would start program execution (does set the ip, Instruction Pointer, to the 257th byte) at our first command, push ax.

This much to the whole simplicity as it was way back then. Of course, there is still a lot of programming going on right now with old DOS. But how come, that a 'P' becomes an assembler command: Simple, the value of the memory location is still 'P', exactly spoken 0x50, and because the ip points to it, and the operating system is programmed such, that it interprets this value 0x50 at this position as a command.

(And of course, because the value 0x50 is assigned the mnemonic 'push ax', and the intel processor is constructed such, that if it gets voltage at the wires from which it gets the 'orders to execute' which can be seen by the 'set and not set' bits 01010000, which corresponds to 0x50, one wire for each bit, roughly spoken.

The processor would then switch the internal electricity flow in the next cycles (current processor speed about 1 to 3 gigabyte cycles a second) at the data bus, etc, to write the value of ax to the stack, very roughly spoken. - register, stack, etc, see lesson two.

And my last comment within brackets - one of the first processors were called 8088, meaning 8 wires data bus and a 8 wires for the commands, the following, next generation process was called 8086, 16 bit data bus.)

Now, in case you do not understand any more, why 0x50 is shown as 'P' in your text editor, any ASCII table will help, I guess you can find it with 'symbol table' in your windows help.

same value has different meanings

I did of course use 'P' on purpose. This very value at this exact position can still be interpreted, as a 'P', which you can see later in this lesson. To make this clear again: notepad or any text editor reads the value 0x50 at the first position of the text file - rather rundos00.com file opened as text file - as 'P', the same file opened from the command prompt by DOS reads this 0x50 as 'push ax' and does not see it as the first, but the 257th byte.

You should not run rundos00.com on your xp cmd prompt, unless you like to risk a crash. Our following simple program can be run at xp cmd, an endless loop, because it can be 'properly' closed with the windows task manager.

But first let us see, if the 'P' is still there in our rundos00.com, as seen by DOS, and where it is. (If you did get familiar with AFD, you can verify this all yourself.)

Taking a full look at AFD,

pdfcrowd

```
AFD.EXE
AX 0000   SI 0000   CS 1DB2   IP 0100     Stack +0 0000  Flags 3202
BX 0000   DI 0000   DS 1DB2               +2 20CD
CX 0013   BP 0000   ES 1DB2   HS 1DB2     +4 9FFF  OF DF IF SF ZF AF PF CF
DX 0000   SP FFFE   SS 1DB2   FS 1DB2     +6 9A00   0  0  1  0  0  0  0  0

CMD >

0100 50            PUSH   AX
0101 50            PUSH   AX
0102 50            PUSH   AX
0103 50            PUSH   AX
0104 50            PUSH   AX
0105 50            PUSH   AX
0106 50            PUSH   AX
0107 50            PUSH   AX

1          0  1  2  3  4  5  6  7
DS:0000   CD 20 FF 9F 00 9A F0 FE
DS:0008   1D F0 1B 05 58 0A 4B 01
DS:0010   EE 03 56 01 EE 03 25 05
DS:0018   01 01 01 00 02 FF FF FF
DS:0020   FF FF FF FF FF FF FF FF
DS:0028   FF FF FF FF 7E 1D C0 11
DS:0030   35 05 14 00 18 00 B2 1D
DS:0038   FF FF FF FF 00 00 00 00
DS:0040   05 00 00 00 00 00 00 00
DS:0048   00 00 00 00 00 00 00 00

2          0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
DS:0000   CD 20 FF 9F 00 9A F0 FE   1D F0 1B 05 58 0A 4B 01
DS:0010   EE 03 56 01 EE 03 25 05   01 01 01 00 02 FF FF FF
DS:0020   FF FF FF FF FF FF FF FF   FF FF FF FF 7E 1D C0 11
DS:0030   35 05 14 00 18 00 B2 1D   FF FF FF FF 00 00 00 00
DS:0040   05 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

1 Step  2ProcStep  3Retrieve  4Help ON  5BRK Menu  6  7 up  8 dn  9 le  10 ri
```

we notice, that we can possibly see our 'P', when we set the data segment to a better position.

```
C:\DOKUME~1\ben\Desktop\AFD.EXE
AX 0000   SI 0000   CS 1DB2   IP 0100     Stack +0 0000  Flags 3202
BX 0000   DI 0000   DS 1DB2               +2 20CD
CX 0013   BP 0000   ES 1DB2   HS 1DB2     +4 9FFF  OF DF IF SF ZF AF PF CF
DX 0000   SP FFFE   SS 1DB2   FS 1DB2     +6 9A00   0  0  1  0  0  0  0  0

CMD >

0100 50            PUSH   AX
0101 50            PUSH   AX
0102 50            PUSH   AX
0103 50            PUSH   AX
0104 50            PUSH   AX
0105 50            PUSH   AX
0106 50            PUSH   AX
0107 50            PUSH   AX

1          0  1  2  3  4  5  6  7
DS:0000   CD 20 FF 9F 00 9A F0 FE
DS:0008   1D F0 1B 05 58 0A 4B 01
DS:0010   EE 03 56 01 EE 03 25 05
DS:0018   01 01 01 00 02 FF FF FF
DS:0020   FF FF FF FF FF FF FF FF
DS:0028   FF FF FF FF 7E 1D C0 11
DS:0030   35 05 14 00 18 00 B2 1D
DS:0038   FF FF FF FF 00 00 00 00
DS:0040   05 00 00 00 00 00 00 00
DS:0048   00 00 00 00 00 00 00 00

2          0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
DS:0100   50 50 50 50 50 50 50 50   50 50 50 50 50 50 50 50
DS:0110   50 50 50 00 00 00 00 00   00 00 00 00 00 00 00 00
DS:0120   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
DS:0130   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
DS:0140   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

1 Step  2ProcStep  3Retrieve  4Help ON  5BRK Menu  6  7 up  8 dn  9 le  10 ri
```

our first (continuously) running asm program

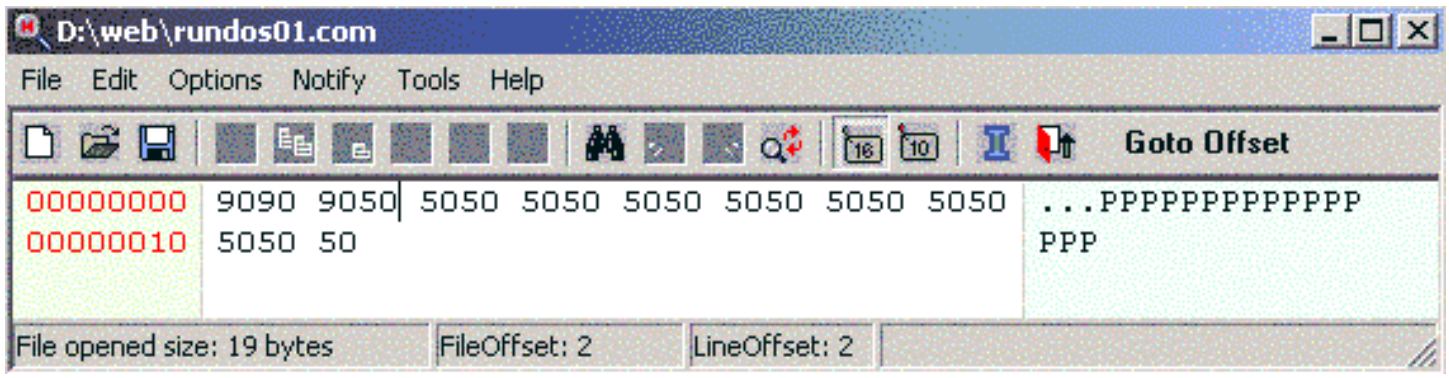
To make it, we rename rundos00.com to rundos01.com and open it

```
D:\web\rundos01.com
File  Edit  Options  Notify  Tools  Help
                                                    Goto Offset
00000000  5050 5050 5050 5050 5050 5050 5050 5050   PPPPPPPPPPPPPPPP
00000010  5050 50                                   PPP
File opened size: 19 bytes   FileOffset: 0   LineOffset: 0
```

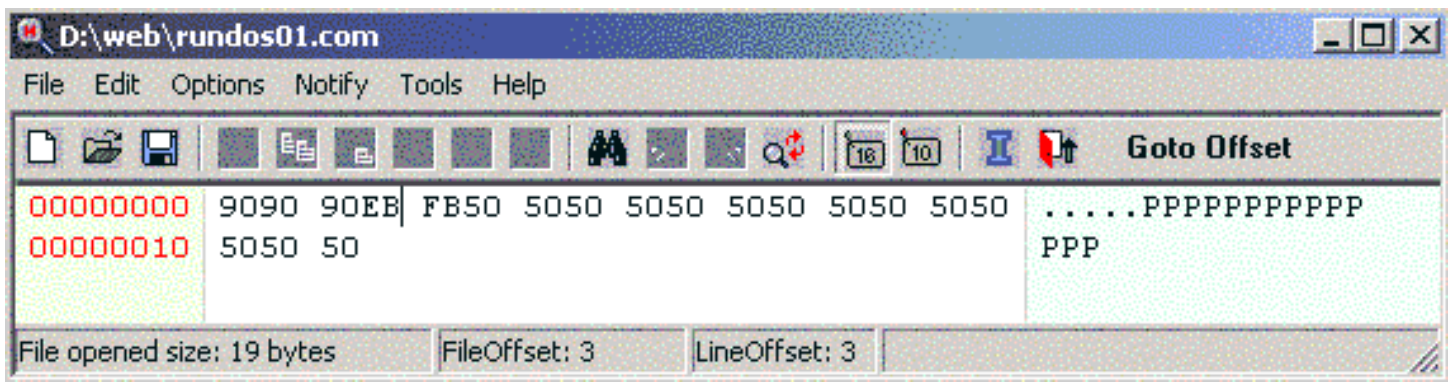
pdfcrowd

in a simple hex editor.

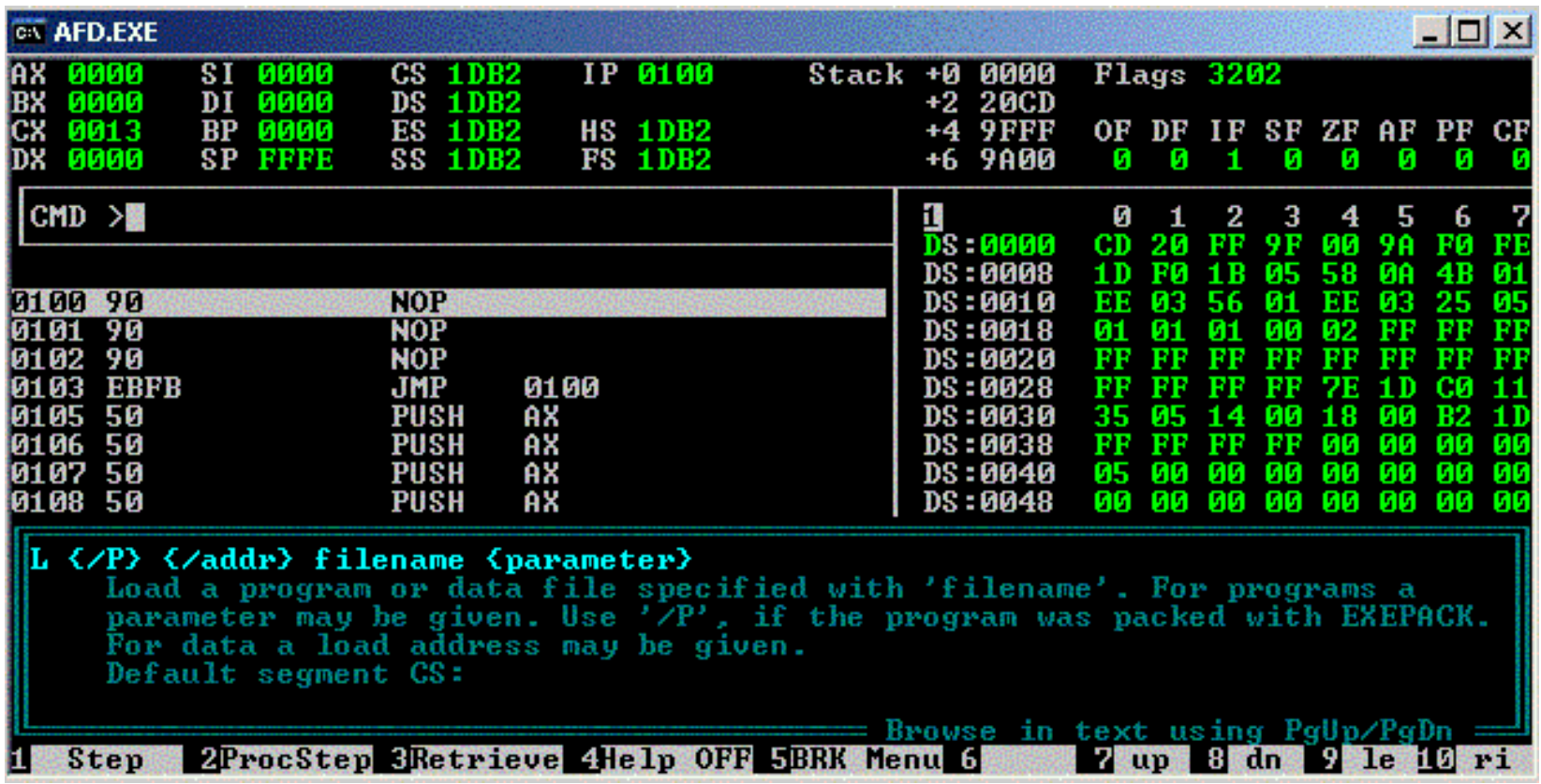
Then we change the first 3 0x50 to 0x90, this corresponds to our known mnemonic 'NOP'.



Next byte we change to 0xeb, and the next to 0xfb.



The two last bytes are understood as one command and correspond to jmp 0100, as can be seen in AFD.



This is all. When we start rundos01.com at our windows cmd prompt, it will run continuously, as it is an endless loop. To make sure, you have it right, here to download. Best way to stop it, close the cmd or use the task manager. Except eating up your processor time, it should do no harm.

Well, of course, this looks a bit shallow, but we do not want to exploit on this too much, because,

pdfcrowd

we do not need to go too much into DOS programming. My point was to show some of the basic principles. You can use AFD if you want to go into details, however, for the understanding the rest of this lesson, you do not need to.

If you want to go into it however, you can solve the following two exercises:
1) see if you can guess the mnemonics push bx, push cx and push dx with just using notepad and see it in AFD,
2) try to make the short sample program - endless loop - even shorter.

and this is the road map for the rest of this lesson

We make the same endless loop as a proper windows program, using flat assembler (FASM) here is the same program's source file for masm, and, and this is the most important 'key', look at it with windbg (which of recent, I could even find at their site, using search: 'windbg'; it was always a free microsoft program) and IDA, the very posh disassembler.

Then you are on your own to study a 'real' windows program, but with a sample written for this purpose and a short explanation of windows messaging, means, how our program gets the message from windows, e.g. so that we know, that the mouse was clicked.

I did also find it difficult to study the mechanism to refresh a display with paint, and how this relates to create, so another two samples in FASM do follow, which also show the interaction with a menu. The fasm samples do not use my favored callback procedure, so they might be easier to study. Again, what is most important to learn: how to use windbg - and I did not find any mention of this ever when I needed it most, learning assembler for windows. If you know how to use IDA and windbg, you can actually debug any program, which you are working on within your favored integrated developing environment.

You do not even need IDA, which is a very expensive program, the version I offer for download is a rare free version, which does not have the internal debugger of IDA activated, and I do not know, how much you can save your notes, e.g. if you want to name the value at 0x00402016 to 'amount for time cycles'; we do not need this for our debugging purposes, though it might make the work easier. Also, I do not go into symbols, but will show you how we can use windbg without their use. (Of course, symbols, just as windbg are free, but a lot of downloading or some installing required).

When I am programming, usually c, and things do need closer attention, most commonly if I want some routines running faster and thus use assembler, I do generally need IDA once, which helps me to locate my code currently worked on. Then, for following changes, windbg is enough. But often enough, e.g. if a new variable or command is inserted in my source code, the starting addresses, where I want to start to look into, do shift slightly: in such a case, a fast and easy to operate disassembler would be more handy than IDA.

I guess the most handy free disassembler is BORG. At this site you can also find a good program to analyze the file header, peinfo, all with source code including.

You can also use ntsd. New Technology Symbolic Debugger is part of every windows xp and 2k. Use just the 'u' unassemble function of this NT follow up debugger of old dos debug. Just type directly, at any cmd prompt, which directory holds your .exe:
ntsd -c "u0401000;" myprog.exe
0x000401000 is the start of the code section in many programs. And if you know, where you code starts, and there are a lot of pe header info analysis programs to find, like pedump or peview, you have all you need, fast and free.
Usually it is difficult to find where to look for a possible error, but most debuggers of commonly

pdfcrowd

used programming language IDE give you some address, which does look something like at 0x00401d32. In such a case as well, a very simple disassembler or just windbg might be enough.

our first (continuously running) windows assembler program

Not as easily 'compiled' as in DOS, we use, however, exactly the same mnemonics, except that we can not jump to 0100, but to a label, and we do need a compiler, means a program, which makes a proper running .exe from our 3 times NOP and one jump. Then we look into runwin01.exe with windbg. Because it is easier to compile, I do discuss this sample code coded for fasm, (here it is for masm).

Just type the console version (fasm.exe, here directly to download)
fasm runwin01.asm
at a suitable cmp prompt and you get runwin01.exe. When we disassemble runwin01.exe, we see

```
00401000
00401000                public start
00401000 start          proc near
00401000                nop
00401001                nop
00401002                nop
00401003                jmp     short start
00401003 start          endp
```

in IDA, that we do have essentially the same program, which we did have with rundos01.com. Well, it makes me a bit considerate, shall I really start this in my windows? What happens, if this wild loop, no precautions or whatever built in, does crash my dirty xp installation? With our .com, we could at least close the command window. Why not first step through it with windbg, just to be save. (And if you are not in the mood to download windbg, you could just do this with ntsd as well, as it is that short, and easy to handle, that this would hardly make any difference for a program with four lines.)

Started windbg and loaded runwin01.exe, we have to type 0401000 at the top of the disassembly window, which we can open at the menu, (and likely have to type this twice)

Disassembly

Offset: 0401000

```
00400fe4 0000              add     [eax],al
00400fe6 0000              add     [eax],al
00400fe8 0000              add     [eax],al
00400fea 0000              add     [eax],al
00400fec 0000              add     [eax],al
00400fee 0000              add     [eax],al
00400ff0 0000              add     [eax],al
00400ff2 0000              add     [eax],al
00400ff4 0000              add     [eax],al
00400ff6 0000              add     [eax],al
00400ff8 0000              add     [eax],al
00400ffa 0000              add     [eax],al
00400ffc 0000              add     [eax],al
00400ffe 0000              add     [eax],al
00401000 90                nop
00401001 90                nop
00401002 90                nop
00401003 ebfb              jmp     image00400000+0x1000 (00401000)
00401005 0000              add     [eax],al
```

and see our code, just like in IDA. It is easy now, click somewhere in the middle of the NOPs

pdfcrowd

```
00400ffc 0000              add     [eax],al
00400ffe 0000              add     [eax],al
00401000 90                nop
00401001 90                nop
00401002 90                nop
00401003 ebfb              jmp     image00400000+0x1000 (00401000)
00401005 0000              add     [eax],al
```

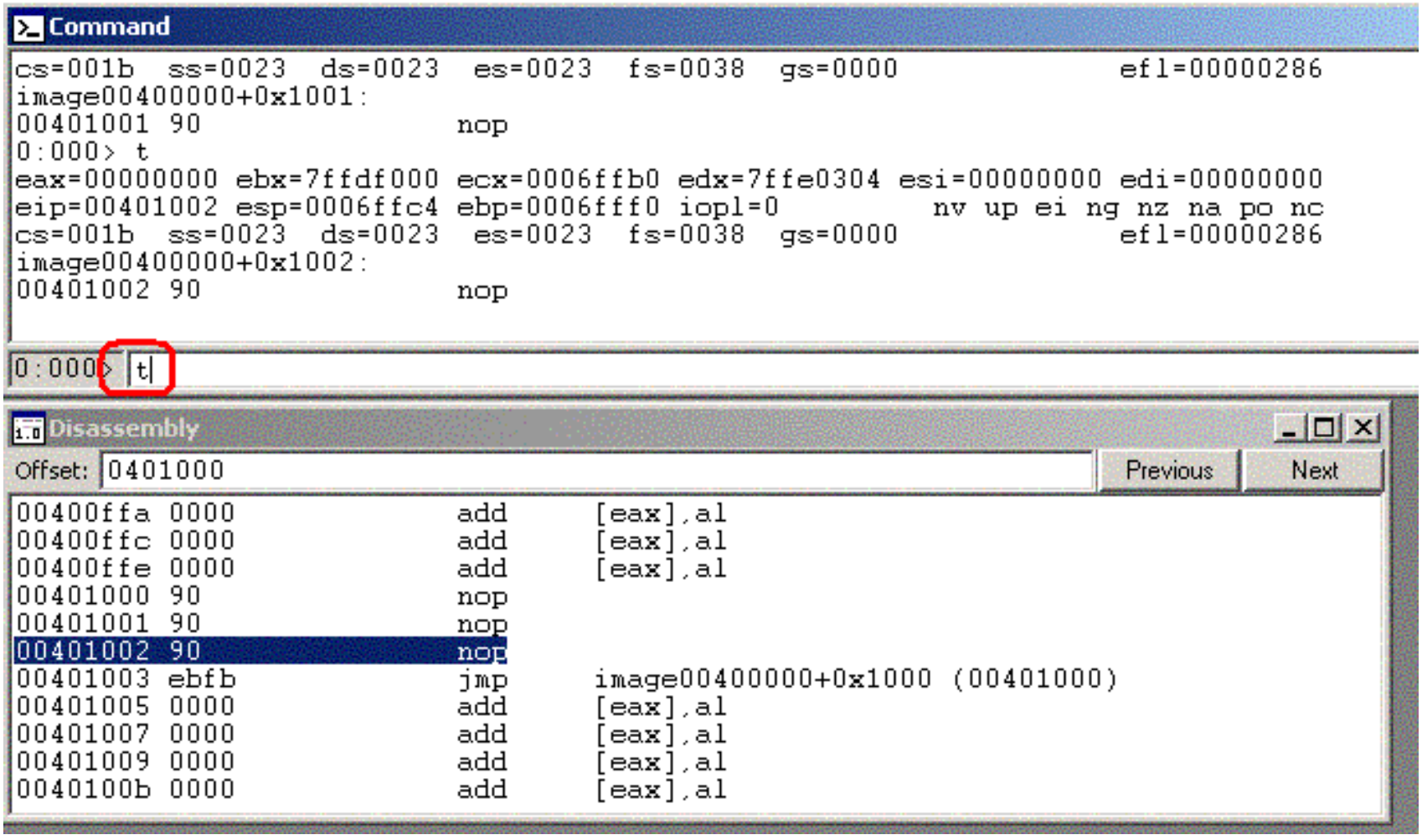
and press the function key for 'run to cursor', e.g. F7

```
Command
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.
ntdll!DbgBreakPoint:
77f6f570 cc                int     3
0:000> g 0x401001
eax=00000000 ebx=7ffdf000 ecx=0006ffb0 edx=7ffe0304 esi=00000000 edi=00000000
eip=00401001 esp=0006ffc4 ebp=0006fff0 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000286
image00400000+0x1001:
00401001 90                nop
0:000>
```

```
Disassembly
Offset: 0401000                                          Previous   Next
00400ff8 0000              add     [eax],al
00400ffa 0000              add     [eax],al
00400ffc 0000              add     [eax],al
00400ffe 0000              add     [eax],al
00401000 90                nop
00401001 90                nop
00401002 90                nop
00401003 ebfb              jmp     image00400000+0x1000 (00401000)
00401005 0000              add     [eax],al
00401007 0000              add     [eax],al
00401009 0000              add     [eax],al
```

which will show you above. The program will be executed to where you clicked. Type 't' repeatedly at the command window, and

pdfcrowd



thus trace and watch execution of the endless loop.

a more challenging debugging job

windows messages and stack frames

It is a bit tricky to catch the windows messages, means to see them, when we get them sent to our program while debugging it. The following sample will demonstrate this in all details, but first a word about stack frames, because we need to understand, why they are necessary in windows, and of course, what they are. Best explanation is still, and can not be topped, by Jeremy Gordon, sorry, you have to find it there yourself. (Also, there is no better introduction: how to learn programming in assembler, the actual act of writing the program, like all the mov eax,ebx, etc. than for his goasm).

The stack is the place in memory, which is shared by all programs, and where information is handed over from one module to an other, just because it is shared. When windows calls our program, it does, between others, two things: it gives permission to us to execute (run our program from our ip, Instruction Pointer), and does let us know, where we can find the messages, which are sent continuously from windows to our program, while it is running.

These messages must be read by our program, and usually worked off, one by one. When the mouse is clicked, we react on this, usually working off some part of the program. When this is done, we want to know if we got a key pressed or if the mouse is clicked again. And of course, these messages are written, by windows, at memory locations of the stack, from where we can read them.

(Good reference for most messages comes with www.eji.com, the example for windows. But, all information you ever need are free from microsoft, e.g. search SDK, but be prepared for a lot of

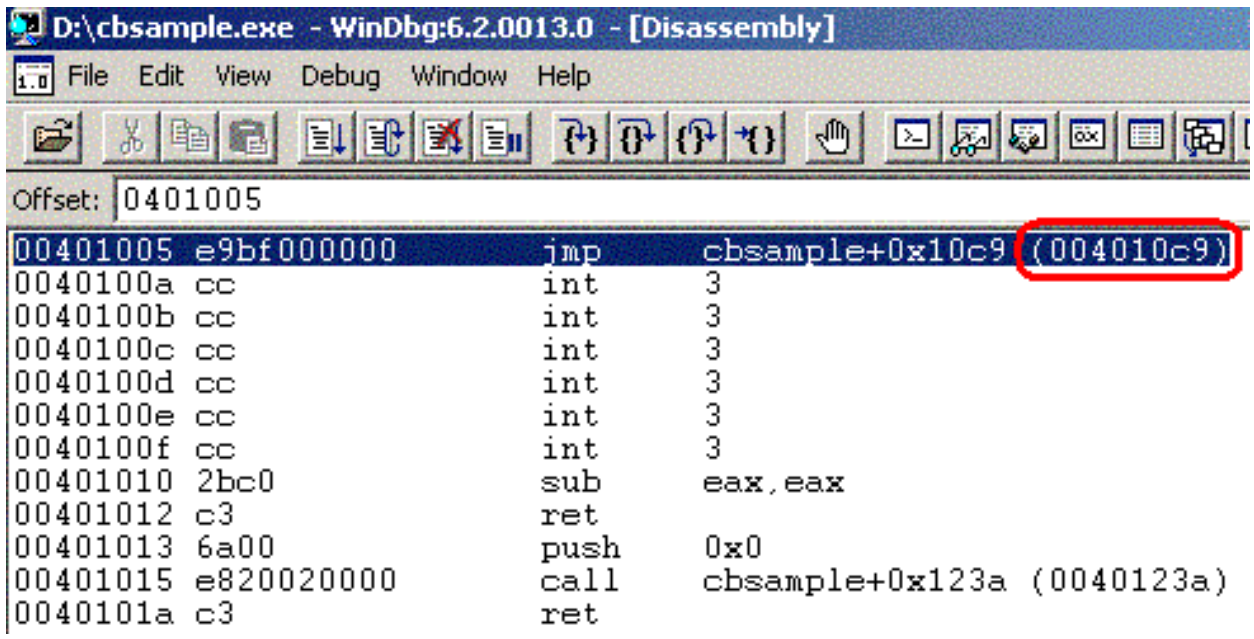
pdfcrowd

digging before you get anywhere.)

So, why do we need a stack frame, and what is it. It is a save place on the stack. Windows needs this, because, very roughly explained (there are may other reasons for a stack frame, especially programming c) it happens routinely, that our program is called by windows, and while we work off the message from windows, which we just have read from the stack, we do call routines of windows from our program, so that we do get recursively called from windows, which could mess up reading and executing.

This is of course very extremely oversimplified, but essentially does explain it, why windows needs a save space on the stack. The method to make a stack frame is simple. We push data on the stack, and then we add the amount of space we use for our data to the stack pointer. So, all subsequent stack operations do not touch our save area of the stack. We would not be able now to find our data again, if we did not save the information, where we did write it on the stack, usually this is done with ebp, extra base pointer, but you do not need to know any of this in detail, to understand cbsample.exe.

find the program section for evaluating windows messages



As you can see, if we disassemble cbsample.exe with BORG, our code entry is 1005 and the first command is a jump
jmp 004010c9.

At 004010c9 and the next line we see
push 0x0 and
call cbsample+0x124c (0040124c)

We know from our source code, that this can only be
push 00h
call GetModuleHandleA

Clearly this is followed by
mov [cbsample+0x4000 (00404000)],eax

Looking at the source code, we see after the move of the value of the register eax to the memory location 'hInst' an other move,
mov dword ptr [cbsample+0x400c (0040400c)],0x40108f

pdfcrowd

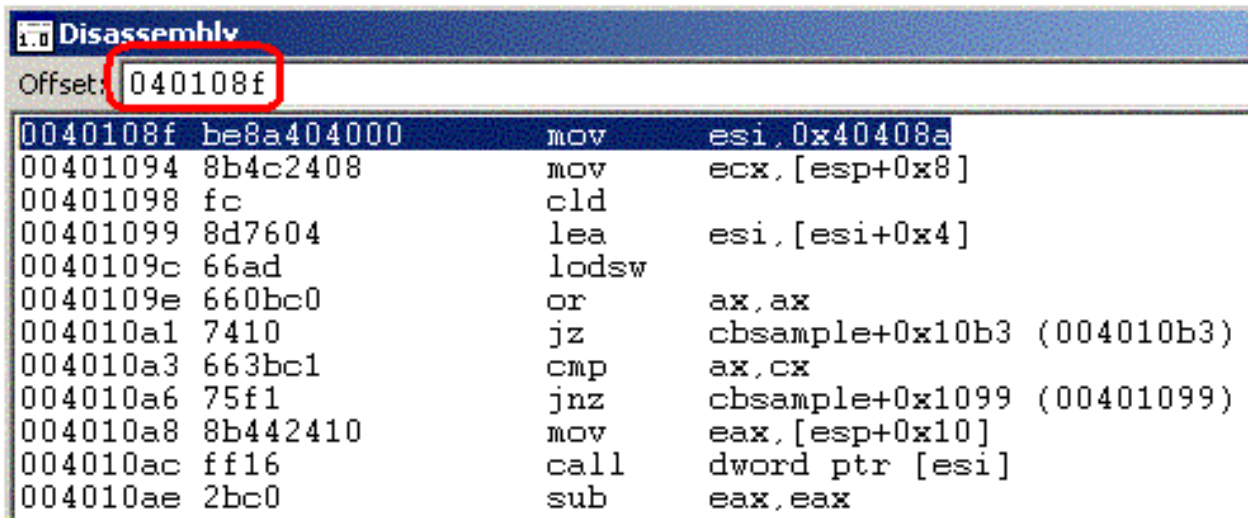
We do not care, to where (memory location: 0040400c) the value 0x40108f shall be moved , but we can see from our source code, what we do move there: it is the starting address of our callback procedure, and this is exactly what we were looking for, because the callback procedure looks at the incoming messages from windows and evaluates them in as much, as it decides, if there shall be any action or not. Our intention is to see when the 0x01 for create comes in, and then we want to see how we branch to our 'create: subroutine' in our code.

But slowly, of course I did not explain jet, what a call to 'GetModuleHandleA' means, how we can look up all these subroutines, why we had to push 0 first, and why we have a value in eax, which we apparently want to keep.

The answer lies in win32 help, (Windows API documentation, about 10 mb and can be downloaded from several places in the net) where we can look up any API. GetModuleHandle does what the name says, and to get such a handle, we have to push a 0x00 at the stack, before we call GetModuleHandle. It takes the 0 from the stack and knows, because it was a 0, that we need a handle for a module, and when it returns control to our program, the value of such a handle is in eax. This is all explained in the various documentation and presented in short form in our win32 help.

Not explained is the 'A' at the end of GetModuleHandle. It stands for ASCII, and can be replaced by 'W', which means wide and represents unicode characters. (An exercise for you for later would be, to write Hello World in Arabic, Hebrew and all the other foreign languages, you are familiar with, using this 'W').

As we do not, at this point, care about anything than finding our messages, which we get from windows, we are just interested in our callback procedure and how we can see it in action, using windbg and stepping through it.



Note, the disassembly in windbg starts at 0040108f.
mov esi,0x40408a
mov ecx,[esp+0x8]
cld
lea esi,[esi+0x4]
lodsw
or ax,ax
Comparing this with our source code, we can see that,
mov ecx,[esp+8] ;{ gets mssg number from stack
the message number should be in ecx. So, we put the cursor at any place of the line
0040109e 660bc0 or ax,ax

pdfcrowd

```
Disassembly
Offset: 040108f
0040108f be8a404000      mov     esi,0x40408a
00401094 8b4c2408        mov     ecx,[esp+0x8]
00401098 fc              cld
00401099 8d7604          lea     esi,[esi+0x4]
0040109c 66ad            lodsw
0040109e 660bc0          or      ax,ax
004010a1 7410            jz      cbsample+0x10b3 (004010b3)
004010a3 663bc1          cmp     ax,cx
004010a6 75f1            jnz     cbsample+0x1099 (00401099)
004010a8 8b442410        mov     eax,[esp+0x10]
004010ac ff16            call    dword ptr [esi]
004010ae 2bc0            sub     eax,eax
```

and with F7 we run our program to this point. (Windows, through our debugger did start our program, did all the necessary initiating, as required by windows, and did run our program to the spot in our code, where we put the cursor, within our callback procedure, which looks, if the message we get from windows, is such one, which we shall pick up and handle, or if we let windows do the default handling.)

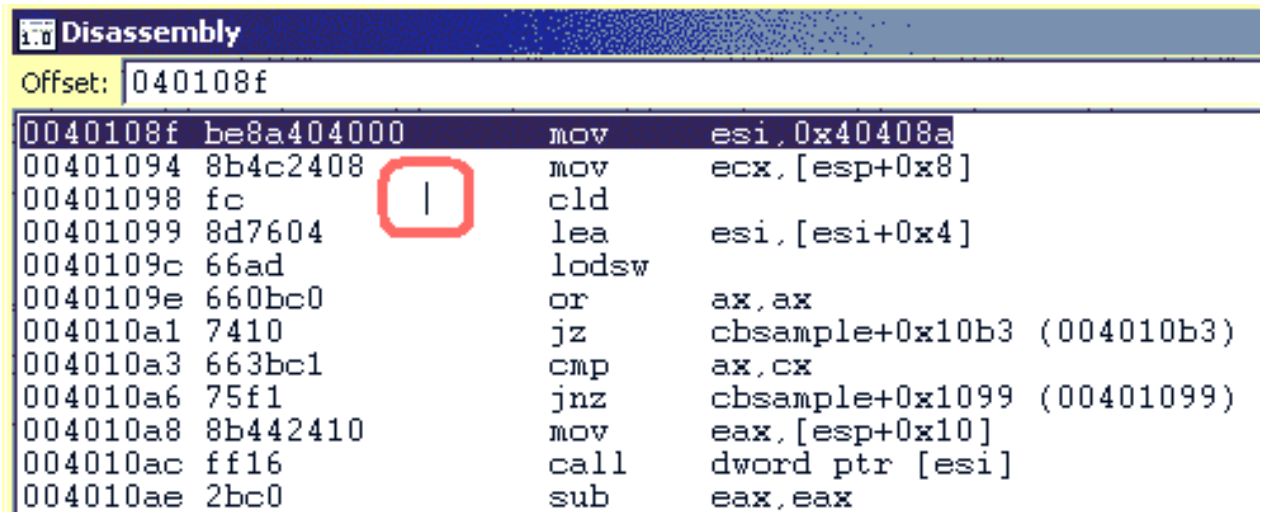
```
Command
ModLoad: 77c90000 77d05000   C:\WINDOWS\system32\RPCRT4.dll
(5f0.408): Break instruction exception - code 80000003 (first chance)
eax=77fb642f ebx=7ffdf000 ecx=00000004 edx=77f67010 esi=00241eb4 edi=00241eb4
eip=77f6f570 esp=0012fb38 ebp=0012fc2c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll
ntdll!DbgBreakPoint:
77f6f570 cc              int     3
0:000> g 0x40109e
ModLoad: 746a0000 746eb000   C:\WINDOWS\System32\MSCTF.dll
eax=c0000001 ebx=00000000 ecx=00000024 edx=00000012 esi=00404090 edi=0012f8f4
eip=0040109e esp=0012f884 ebp=0012f8ac iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000207
cbsample+0x109e:
0040109e 660bc0          or      ax,ax
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WI
0:000>
```

Looking at ecx in the above picture

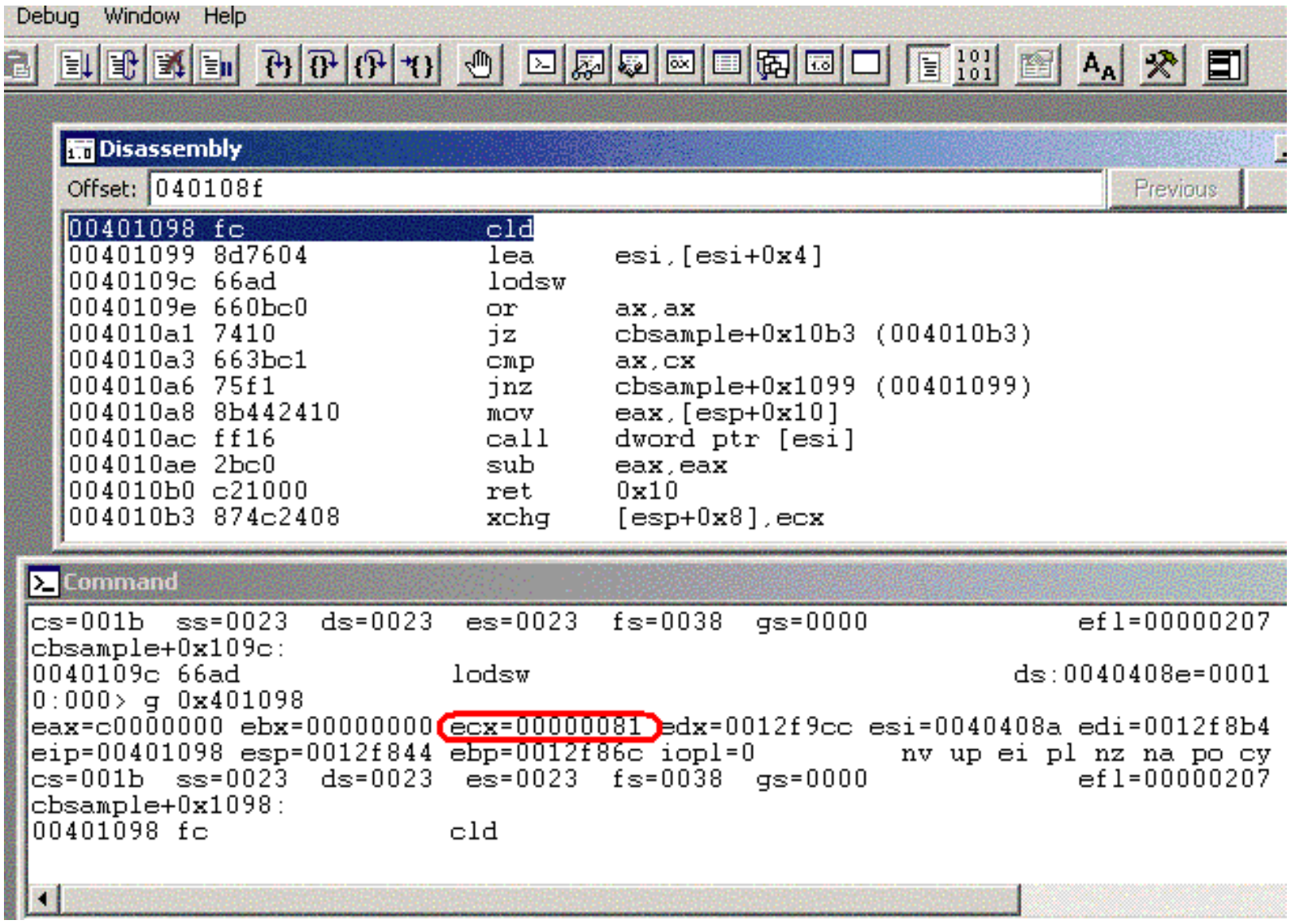
```
Command
ModLoad: 77c90000 77d05000   C:\WINDOWS\system32\RPCRT4.dll
(5f0.408): Break instruction exception - code 80000003 (first chance)
eax=77fb642f ebx=7ffdf000 ecx=00000004 edx=77f67010 esi=00241eb4 edi=00241eb4
eip=77f6f570 esp=0012fb38 ebp=0012fc2c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll
ntdll!DbgBreakPoint:
77f6f570 cc              int     3
0:000> g 0x40109e
ModLoad: 746a0000 746eb000   C:\WINDOWS\System32\MSCTF.dll
eax=c0000001 ebx=00000000 ecx=00000024 edx=00000012 esi=00404090 edi=0012f8f4
eip=0040109e esp=0012f884 ebp=0012f8ac iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000207
cbsample+0x109e:
0040109e 660bc0          or      ax,ax
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WI
0:000>
```

pdfcrowd

we see the value 0x 0000 0024 in ecx. So this was not 0x01 jet. Easiest way to continue: click cursor a few lines above



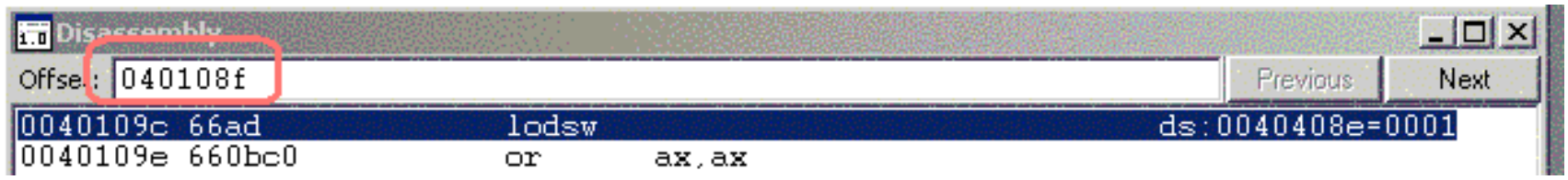
and F7 again and,



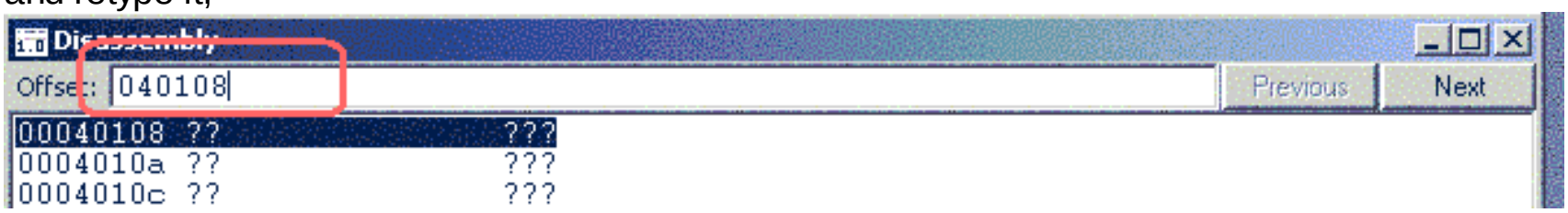
see a new value of ecx. Apparently, there are a lot of spurious windows messages, instead of our CREATE (0x01), but we will continue this long enough, to get the right message. Type one or two 't' at the command window, put the cursor above in the line with 'cld' and F7 and there we are again with a new value in ecx.

If you can not place the cursor in a line above, it is fastest to delete the f in the input line of the disassembly,

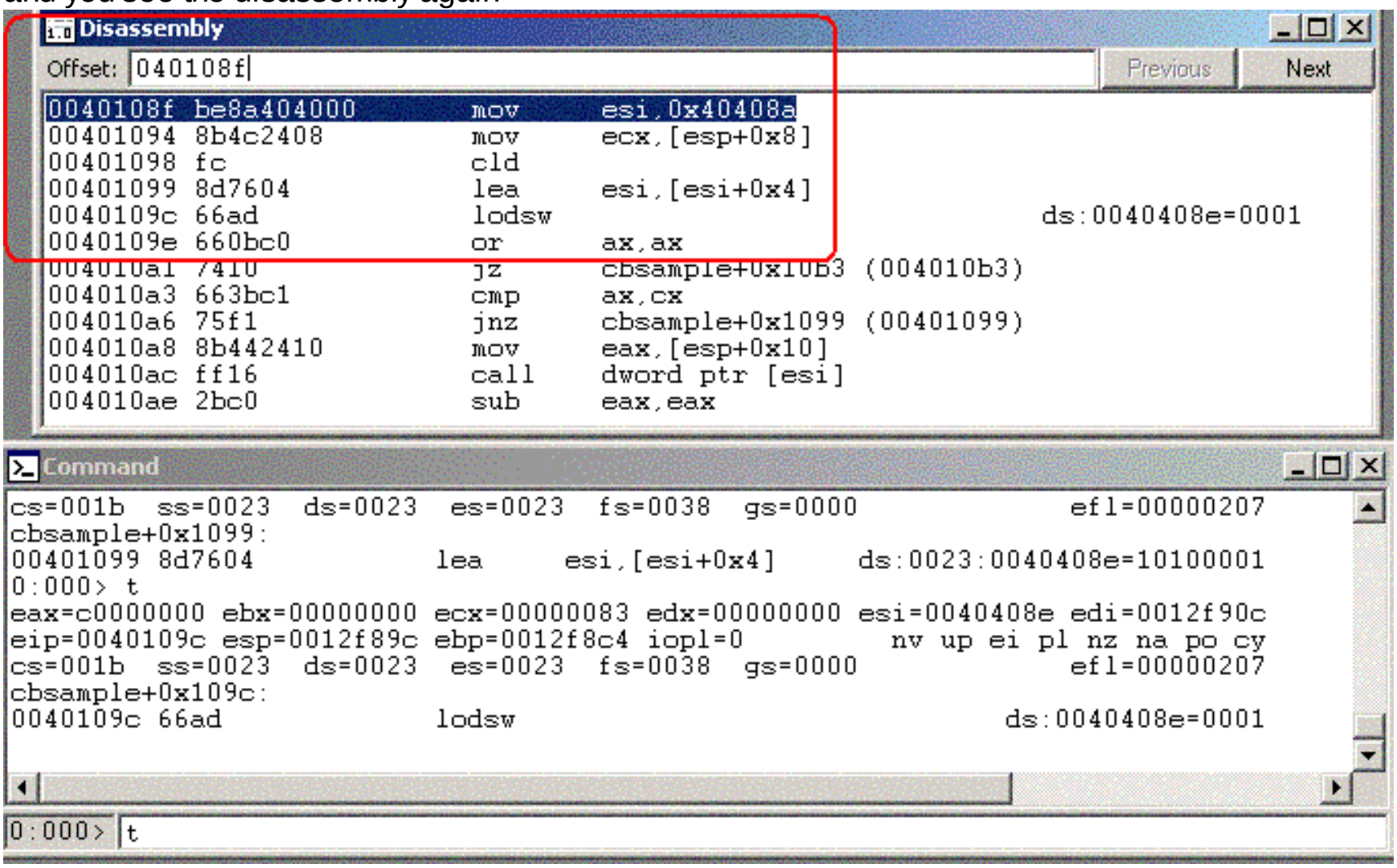
pdfcrowd



and retype it,



and you see the disassembly again



form 040108f onward.

We did run it around, checking the new message, but again, not the one we expect. Reproducing it now it took me 5 times, (in some cases, and possibly awaiting other messages, it has taken me up to 20 times and more,) viola, here we go.

pdfcrowd

Disassembly

Offset: 040108f Previous

```
00401098 fc              cld
00401099 8d7604          lea     esi,[esi+0x4]
0040109c 66ad            lodsw
0040109e 660bc0          or      ax,ax
004010a1 7410            jz      cbsample+0x10b3 (004010b3)
004010a3 663bc1          cmp     ax,cx
004010a6 75f1            jnz     cbsample+0x1099 (00401099)
004010a8 8b442410        mov     eax,[esp+0x10]
004010ac ff16            call    dword ptr [esi]
004010ae 2bc0            sub     eax,eax
004010b0 c21000          ret     0x10
004010b3 874c2408        xchg    [esp+0x8],ecx
```

Command

```
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000207
cbsample+0x109c:
0040109c 66ad            lodsw                              ds:0040408e=0001
0:000> g 0x401098
eax=c0000000 ebx=00000000 ecx=00000001 edx=0012f980 esi=0040408a edi=0012f864
eip=00401098 esp=0012f7f4 ebp=0012f81c iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000207
cbsample+0x1098:
00401098 fc              cld
```

If we now just keep stepping,

Disassembly

Offset: 040108f

```
00401099 8d7604          lea     esi,[esi+0x4]     ds:0023:0
0040109c 66ad            lodsw
0040109e 660bc0          or      ax,ax
004010a1 7410            jz      cbsample+0x10b3 (004010b3)
004010a3 663bc1          cmp     ax,cx
004010a6 75f1            jnz     cbsample+0x1099 (00401099)
```

Command

```
00401099 8d7604          lea     esi,[esi+0x4]     ds:0023:0
```

0:000> t

Disassembly

Offset: 040108f

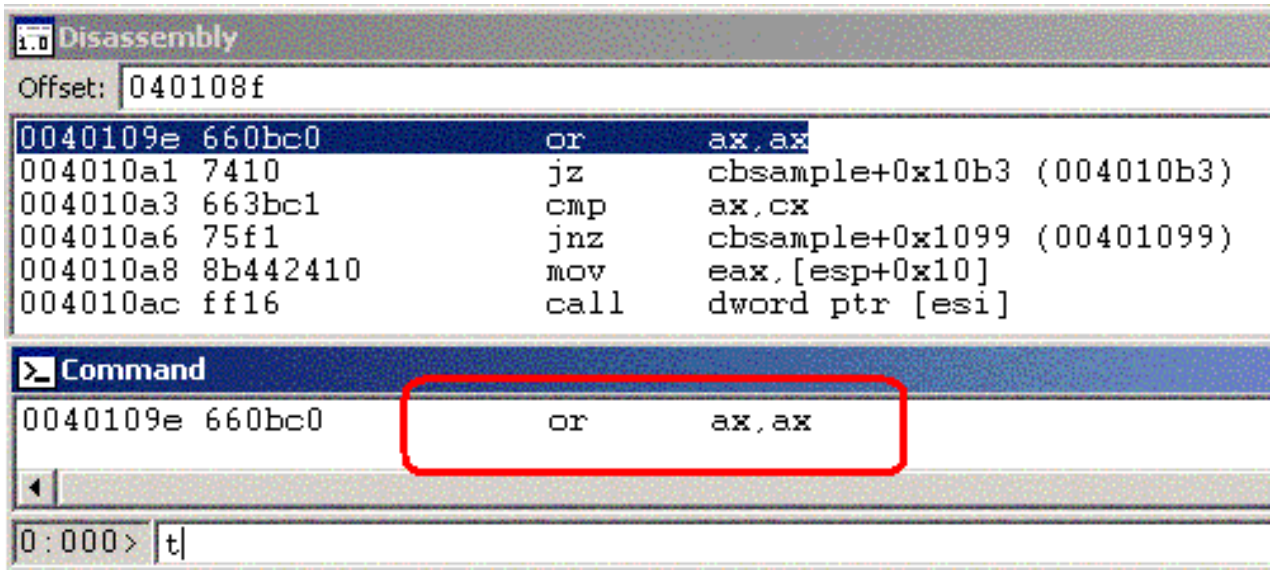
```
0040109c 66ad            lodsw
0040109e 660bc0          or      ax,ax
004010a1 7410            jz      cbsample+0x10b3 (004010b3)
004010a3 663bc1          cmp     ax,cx
004010a6 75f1            jnz     cbsample+0x1099 (00401099)
004010a8 8b442410        mov     eax,[esp+0x10]
```

Command

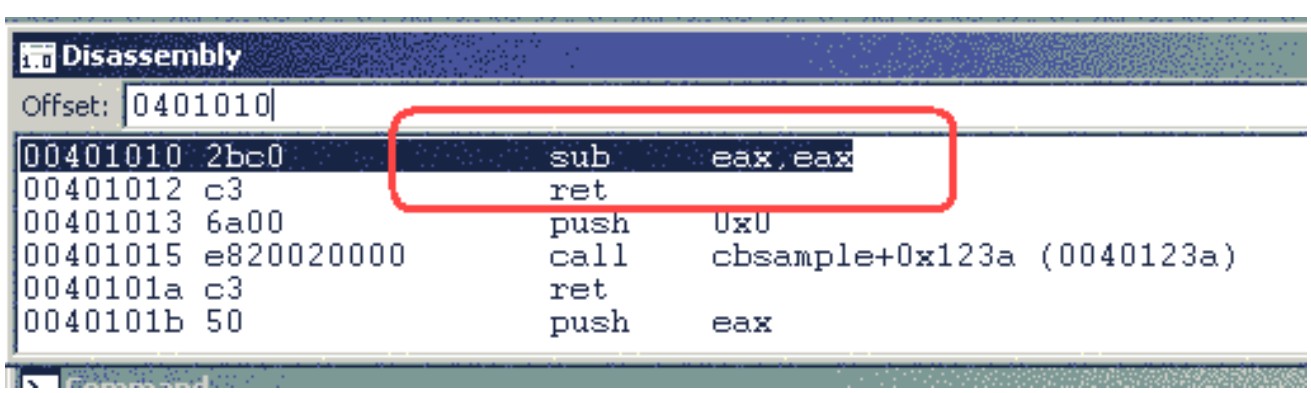
```
0040109c 66ad            lodsw
```

0:000>

pdfcrowd



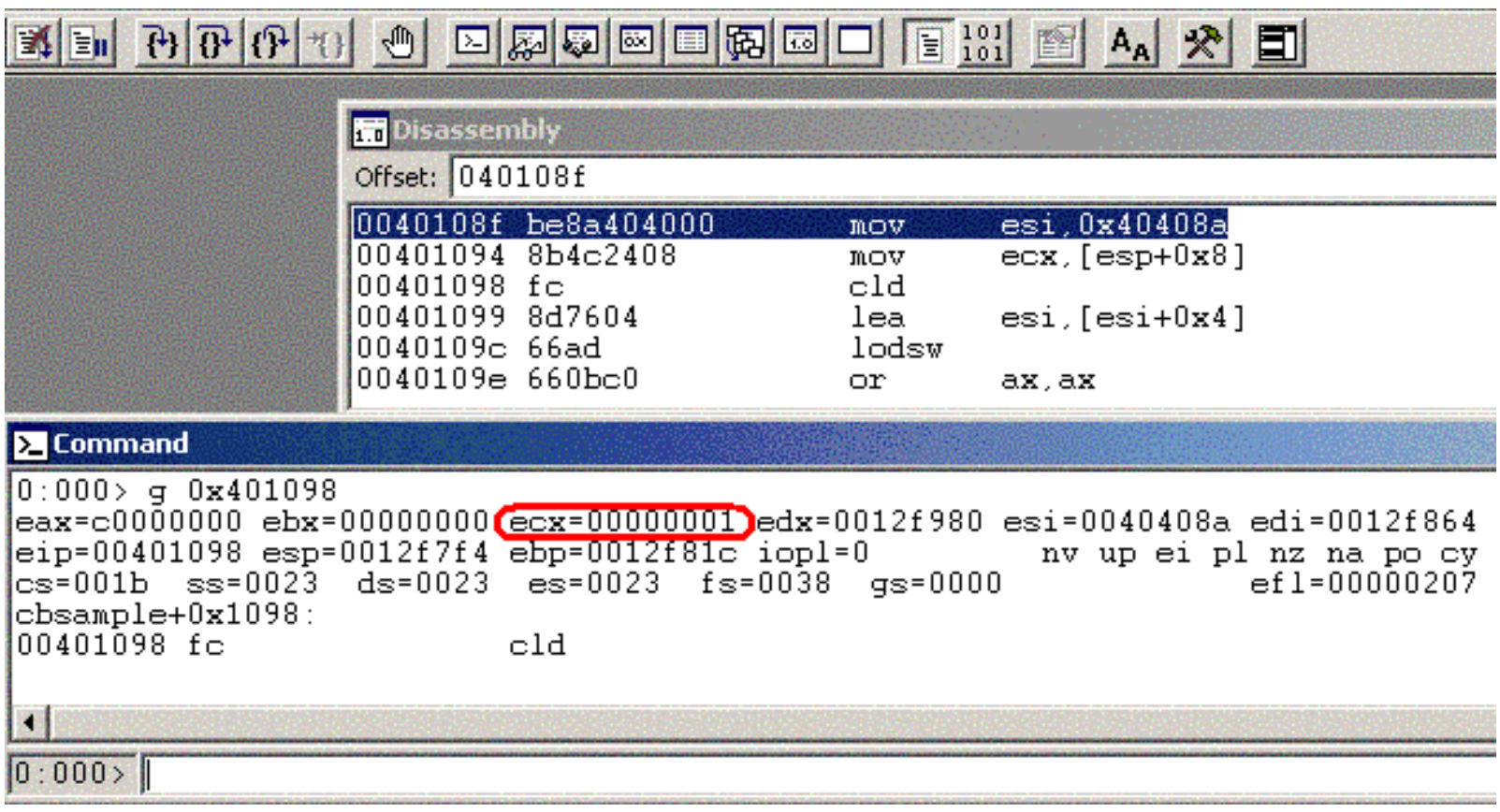
we finally come to



and this is where our 'create:' code would be written, we do only have a simple ret, after we made eax to 0.

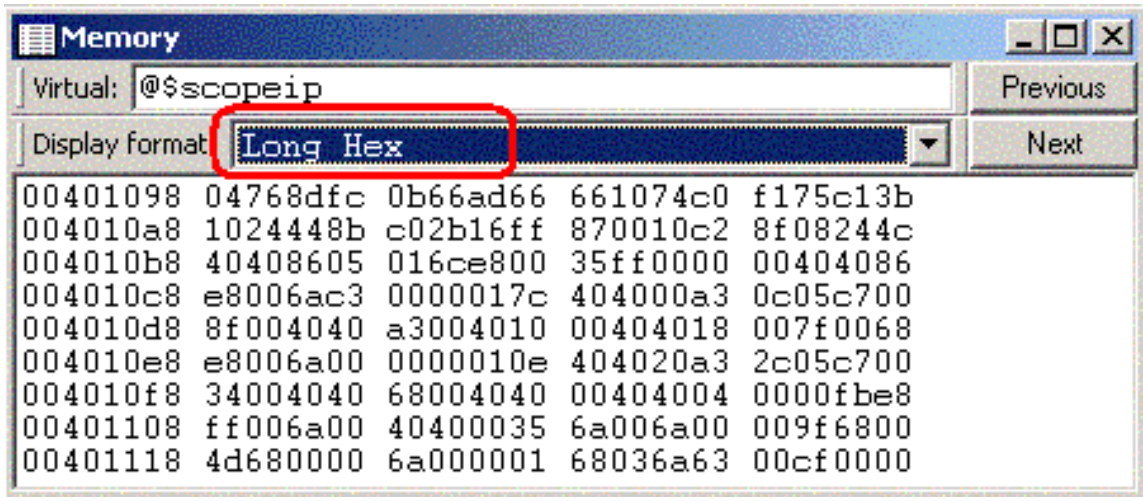
nothing stays hidden using windbg

Next digging deeper is looking directly at the stack, to see, from which position exactly we got our messages. Do the same as above until you get 0x01 message.

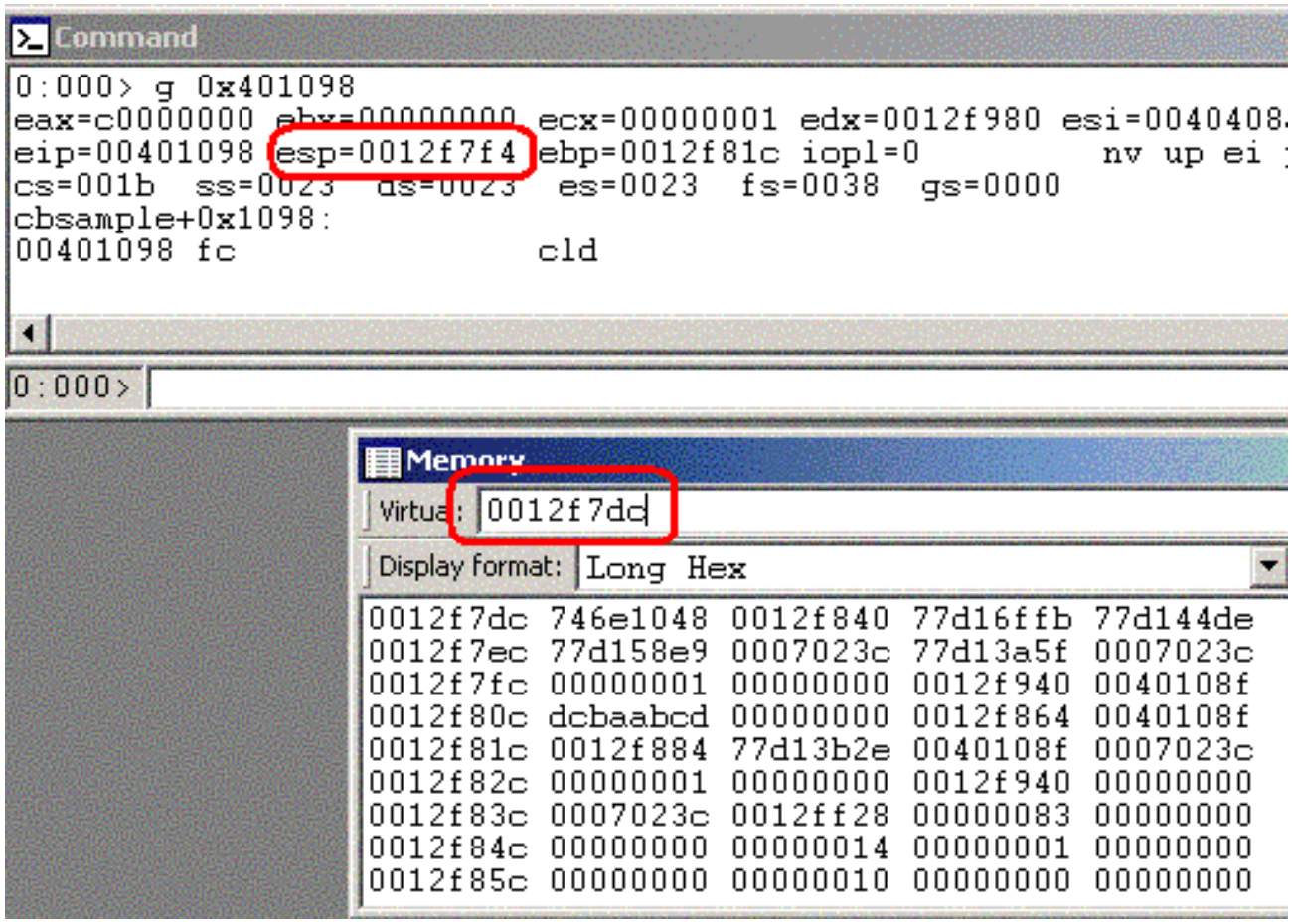


pdfcrowd

Now we open a memory window from the menu of windbg and make it display Long Hex,

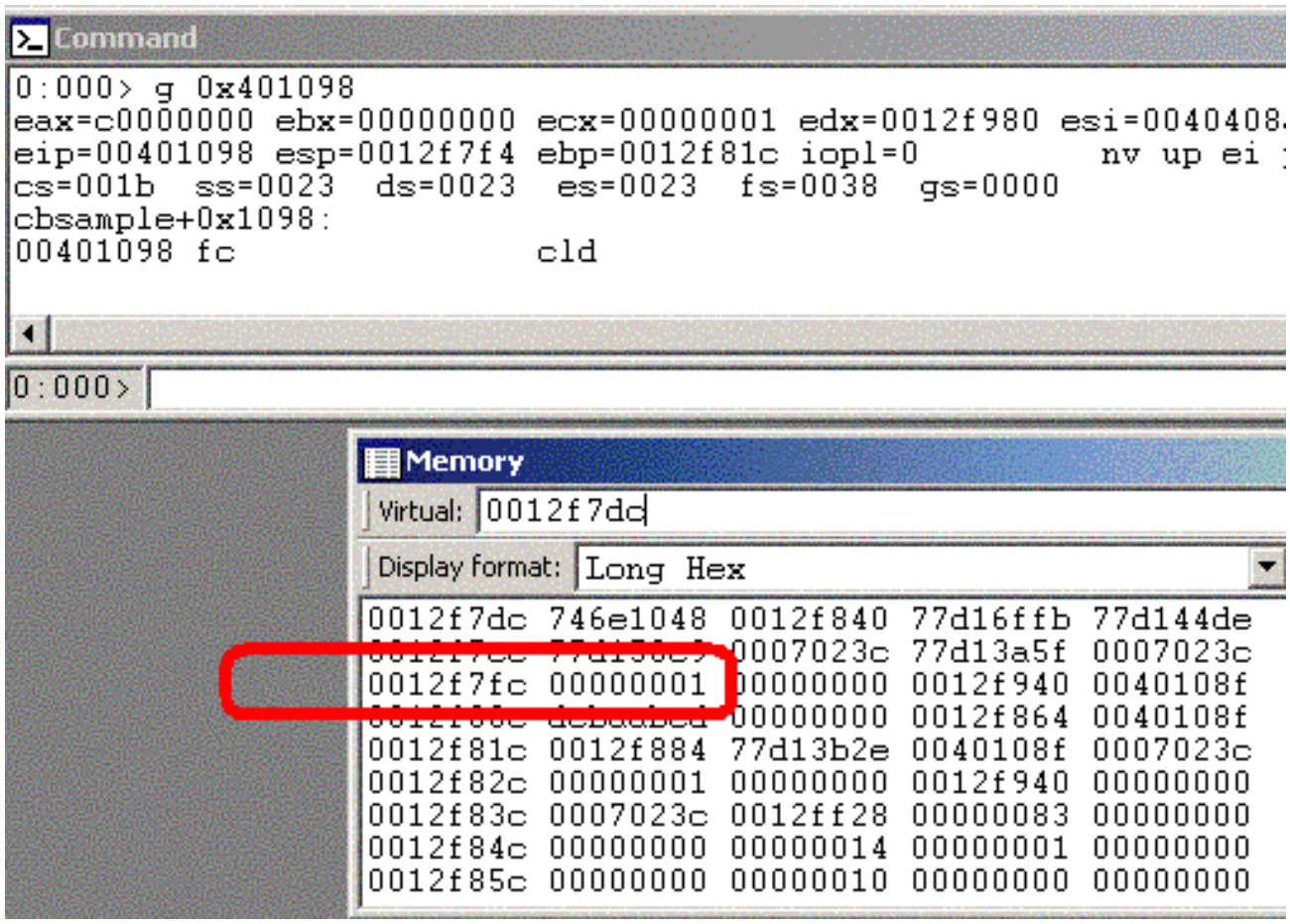


and set it's memory location to about the value of our stack pointer.



And here we can see already our message

pdfcrowd



exactly at the location 0012f7fc. (If we add 8 to 0012f7f4, esp, we do get 0012f7fc - as we did it with the command
'mov ecx,[esp+8]' - and this is how we did take the message number from the stack frame created by windows. We know, that it is 8 added to the stack because we, better our callback procedure, starting at 0040108f, was given control by the API DispatchMessage; when we did register the class with RegisterClassEx, we made the address of our callback procedure known in the data needed to register a class. And all our hacking and possibly some documentation of microsoft, has never shown any other positions of these values on the stack, right after being called by DispatchMessage.)

Once the mechanism is understood, clearly there is nothing more simple than debugging a program. The only real barrier is the huge amount of numbers to remember and having to do it all by hand. This far could be already enough of information, and learning and training material, to put you into the situation, where you can debug programs from other languages with windbg (without any symbols installed). However, it might be a bit of steep learning. So I think it will help, if you can study all mechanisms mentioned above with a simple sample, of course, I recommend cbsample.exe, but leaving out the details of the callback procedure, as it is a fairly simple mechanical matter, which can be reproduced just by some hacking without having more information.

Please note, if you want to use my 386xp callback procedure, you should not use cbsample.asm, as this runs just exactly as is, use this version, fasm instead, which saves esi additionally, which usually has to be done when expanding this sample. Here, masm an other, even 'more save' version, where all registers are preserved, as usually done, by standard 'IF' tables, but unnecessarily so in my opinion.

the basic structure of a windows .exe

We know already, that we first need a handle. Once we have it, windows needs some information to continue with our program. As mentioned above, a class has to be created, by 'call RegisterClassEx'. Properties of such a class are manyfold, and can best we studied with win32 help. Note, you do not see anything jet, but apparently because the windows class needs the

pdfcrowd

address of a callback procedure, it does already send messages to this class. All messages do have the same format, consisting of 7 times four bytes, or 7 times a value of an extended register, like eax.

The second value of a message is the number, which tells us, what the message is about: 0x01 tells the windows class, that it shall execute the code, which we have written in 'create:', in our cbsample, which is just a return. Note, that windows can call this, before it executes further: call CreateWindowEx.

Let me expand a moment here: If you would, for example, want to get the coordinates of your window, which you create with the call CreateWindowEx, perhaps by trying to catch the message 0x05 (get_Size) or whatever method, while in your 'create:', this will fail, because the window was not created jet. (See also further down this lecture.)

After this creation, we have to call the API GetMessage, then it has to be further worked over with TranslateMessage and DispatchMessage. This last DispatchMessage gives control to our callback procedure.

As in the example above, we would come to where we put our cursor in the callback procedure, because this procedure was called by windows, because we did call DispatchMessage. This means, that when windows returns from our call to DispatchMessage, we have worked off all the code called for by our callback procedure, and we are ready to GetMessage again, etc, until the return value of GetMessage is 0x00, at which point we jump out of the loop to our ending routine. The standard way to do this is to call, form our program, for windows to post us this ending message with call PostQuitMessage.

Of course, this all seems to be quite 'thinking around the corner', however, it seems to be a winning approach.

So, wish you fun with studying samples: PAINT is called in addition to CREATE, better via CREATE. We do regularly call PAINT as an automatic call attached to other calls as can be seen here, and this shows you, how you can use the windows edit functions for your programs. One of my favorite to study was Connect from Ron Thomas, and never let us forget Iczelion.

I claim no copyright for myself on this lecture, except that you keep it free, distribute, quote and use any way you want. *IDA, More Properties and windbg, as well as the two additional programs for filedump are not included in the download of the full lecture, because I did want to keep this download small. Very likely - as of current, 2010, some searching will have to be done, to find something comparable, e.g for IDA.*

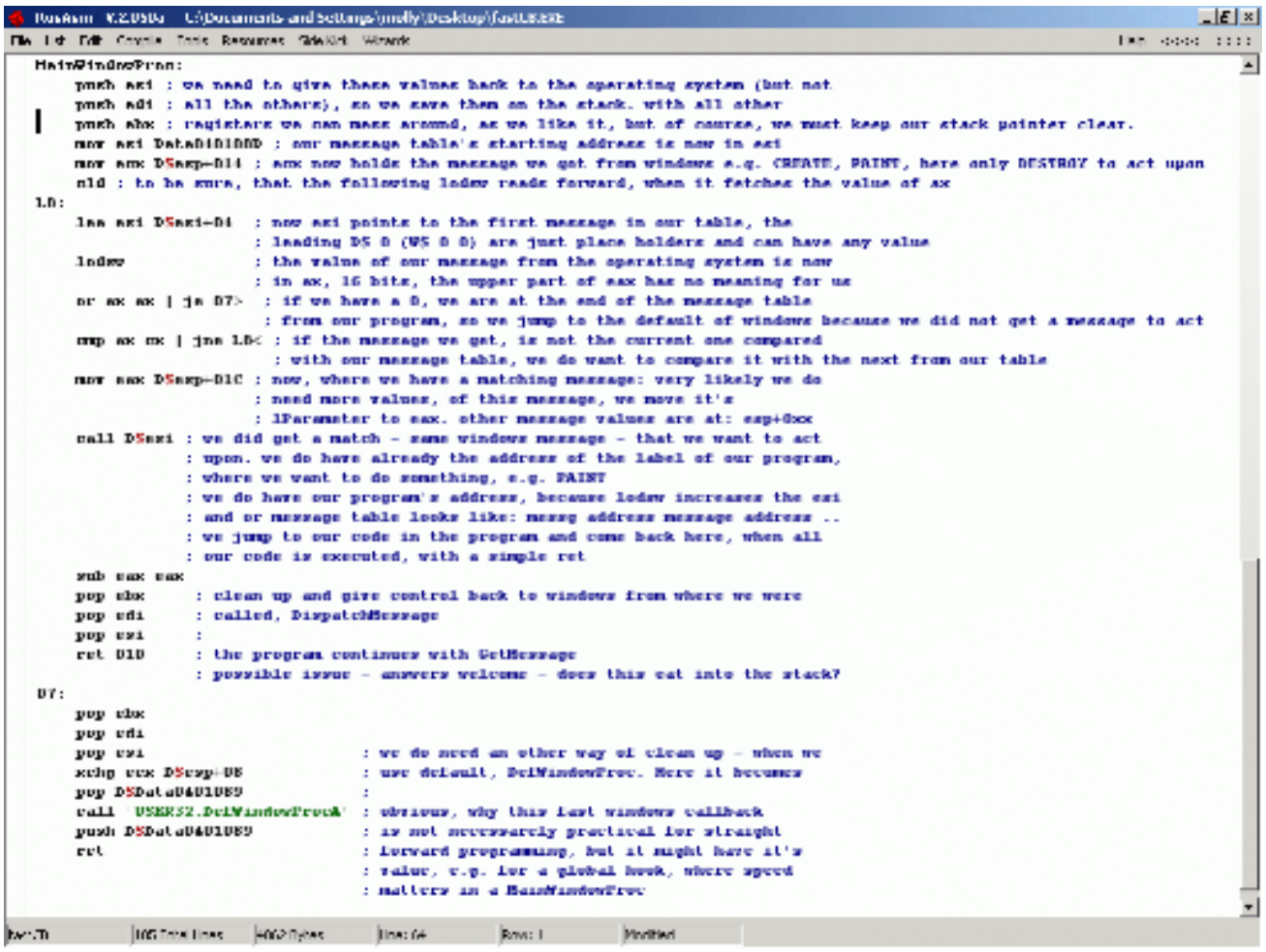
Please note, this tutorial was originally published around 2004, some links will definitely be dead by now, however the essence of asm programming has a sturdy and tough survival attached to it, so it will pay well to search names like Ron Thomas and other. Edits of 2010 in ***italic***

If you are crazy enough or in a special situation, where you actually want or need to use my fast windows callback procedure, here and the following picture

pdfcrowd



is a sample of how to integrate it into c. The pictures serve merely as a documentation, as it is hardly readable and [here]



a pure assembler version. [Rosasm IDE is here]. Note aside, it can be seen easily, even on the two above scratch samples, how much more easy it is to code in asm compared to c for some specific requirements. An other downlod is [here], and in ros forum2 the sample fastCB.zip

Summary of links of tutorial:

pdfcrowd

a86
masm32
MCL, the keyboard applet from mlin.net
FASM
my favored callback procedure
BORG
Jeremy Gordon
Lcc, Jacob Navia
Ron Thomas
Iczelion

direkt weiter mit magie lernen

mit bbogner magie lernen * bbogner magie: sinn oder unsinn * bbogner alte geschaeftsseite * ergaenzungen und downloads

This content belongs to: http://goo.gl/**cmme** - **c**an **m**ake **m**e **e**asy
copyright (c) 2010 Bernat Bogner: free to use but keep it free

pdfcrowd